

15-面向对象之继承：继承是代码复用的合理方式吗？

你好！我是郑晔。

上一讲，我们讨论了面向对象的第一个特点：封装。这一讲，我们继续来看面向对象的第二个特点：继承。首先，你对继承的第一印象是什么呢？

说到继承，很多讲面向对象的教材一般会这么讲，给你画一棵树，父类是根节点，而子类是叶子节点，显然，一个父类可以有多个子类。

父类是干什么用的呢？就是把一些公共代码放进去，之后在实现其他子类时，可以少写一些代码。讲程序库的时候，我们说过，设计的职责之一就是消除重复，代码复用。所以，在很多人的印象中，继承就是一种代码复用的方式。

如果我们把继承理解成一种代码复用方式，更多地是站在子类的角度向上看。在客户端代码使用的时候，面对的是子类，这种继承叫实现继承：

```
Child object = new Child();
```

其实，还有一种看待继承的角度，就是从父类的角度往下看，客户端使用的时候，面对的是父类，这种继承叫接口继承：

```
Parent object = new Child();
```

不过，接口继承更多是与多态相关，我们暂且放一放，留到下一讲再来讨论。这一讲，我们还是主要来说说实现继承。其实，实现继承并不是一种好的做法。

也就是说，**把实现继承当作一种代码复用的方式，并不是一种值得鼓励的做法**。一方面，继承是很宝贵的，尤其是Java这种单继承的程序设计语言。每个类只能有一个父类，一旦继承的位置被实现继承占据了，再想做接口继承就很难了。

另一方面，实现继承通常也是一种受程序设计语言局限的思维方式，有很多程序设计语言，即使不使用继承，也有自己的代码复用方式。

可能这么说你还不太理解，接下来，我就用一个例子来帮你更好地理解继承。

代码复用

假设，我要做一个产品报表服务，其中有个服务是要查询产品信息，这个查询过程是通用的，别的服务也可以用，所以，我把它放到父类里面。这就是代码复用的做法，代码用Java写出来是这样的：

```

class BaseService {
    // 获取相应的产品信息
    protected List<Product> getProducts(List<String> product) {
        ...
    }
}

// 生成报表服务
class ReportService extends BaseService {
    public void report() {
        List<Product> product = getProduct(...);
        // 生成报表
        ...
    }
}

```

如果采用Ruby的mixin机制，我们还可以这样实现，先定义一个模块（module）：

```

module ProductFetcher
    # 获取相应的产品信息
    def getProducts(products)
        ...
    end
end

```

然后，在自己的类定义中，将它包含（include）进来：

```

# 生成报表服务
class ReportService
    include ProductFetcher

    def report
        products = getProducts(...)
        # 生成报表
        ..
    end
end

```

在这个例子中，ReportService并没有继承任何类，获取产品信息的代码也是可以复用的，也就是这里的ProductFetcher这个模块。这样一来，如果我需要有一个获取产品信息的地方，它不必非得是一个什么服务，无需继承任何类。

这是Ruby的做法，类似的语言特性还有Scala里的trait。

在C++中，虽然语法并没有严格地区分实现继承，但《Effective C++》这本行业的名著，给出了一个实用的建议：实现继承采用私有继承的方式实现：

```
class ReportService: private ProductFetcher {  
    ...  
}
```

请注意，在这个实现里，我的私有继承类名是ProductFetcher。是的，它并不需要和这个报表服务有什么直接的关系，使用私有继承，就是为了复用它的代码。

从前面的分析中，我们也不难看出，获取产品信息和生成报表其实是两件事，只是因为生成报表的过程中，需要获取产品信息，所以，它有了一个基类。

其实，在Java里面，我们不用继承的方式也能实现，也许你已经想到了，代码可以写成这样：

```
class ProductFetcher {  
    // 获取相应的产品信息  
    public List<Product> getProducts(List<String> product) {  
        ...  
    }  
}  
  
// 生成报表服务  
class ReportService {  
    private ProductFetcher fetcher;  
  
    public void report() {  
        List<Product> product = fetcher.getProducts(...);  
        // 生成报表  
        ...  
    }  
}
```

这种实现方案叫作组合，也就是说ReportService里组合进一个ProductFetcher。在设计上，有一个通用的原则叫做：**组合优于继承**。也就是说，如果一个方案既能用组合实现，也能用继承实现，那就选择用组合实现。

好，到这里你已经清楚了，代码复用并不是使用继承的好场景。所以，**要写继承的代码时，先问自己，这是接口继承，还是实现继承？如果是实现继承，那是不是可以写成组合？**

面向组合编程

之所以可以用组合的方式实现，本质的原因是，获取产品信息和生成报表服务本来就是两件事。还记得我们在[第3讲](#)里讲过的“分离关注点”吗？如果你能看出它们是两件事，就不会把它们放到一起了。

我还讲过，分解是设计的第一步，而且分解的粒度越小越好。当你可以分解出来多个关注点，每一个关注点就应该是一个独立的模块。最终的**类是由这些一个一个的小模块组合而成，这种编程的方式就是面向组合编程**。它相当于换了一个视角：类是由多个小模块组合而成。

还以前面的报表服务为例，如果使用Java，按照面向组合的思路写出来，大概是下面这样的。其中，为了

增加复杂度，我增加了一个报表生成器（ReportGenerator），在获取产品信息之后，还要生成报表：

```
class ReportService {
    private ProductFetcher fetcher;
    private ReportGenerator generator;

    public void report() {
        List<Product> product = fetcher.getProducts(...);
        // 生成报表
        generator.generate(product);
    }
}
```

请注意，我在前面的表述中，故意用了模块这个词，而不是类。因为ProductFetcher和ReportGenerator只是因为用的是Java，才写成了类；如果用Ruby，它们的表现形式就会是一个module；而在Scala里，就会成为一个trait。我们再用Ruby示意一下：

```
class ReportService
  include ProductFetcher
  include ReportGenerator

  def report
    products = getProducts(...)
    # 生成报表
    generateReport(products)
  end
end
```

而使用C++的话，表现形式则会是有继承：

```
class ReportService: private ProductFetcher, private ReportGenerator {
    ...
}
```

C++本身支持宏定义，所以，我们可以自定义一些宏，将这些不同的概念区分开来：

```
#define MODULE(module) class module
#define INCLUDE(module) private module
```

上面的类定义就可以变成更有表达性的写法：

```
MODULE(ProductFetcher) {
```

```
...
}

MODULE(ReportGenerator) {
    ...
}

class ReportService:
    INCLUDE(ProductFetcher),
    INCLUDE(ReportGenerator) {
        ...
    }
}
```

我有一个C++的高手朋友，把这种做法称之为“小类大对象”，这里面的小类就是一个一个的模块，而最终的大对象是最终组合出来的类生成的对象。

关于面向对象，有一点我们还没有说，就是**面向对象面向的是“对象”，不是类**。很多程序员习惯把对象理解成类的附属品，但在Alan Kay的理解中，对象本身就是一个独立的个体。所以，有些程序设计语言可以直接支持在对象上进行操作。

还是前面的例子，我想给报表服务增加一个接口，对产品信息做一下处理。用Ruby写出来会是这样：

```
module ProductEnhancer
  def enhance
    # 处理一下产品信息
  end
end

service = ReportService.new
# 增加了 ProductEnhancer
service.extend(ProductEnhancer)

# 可以调用 enhance 方法
service.enhance
```

这样的处理只会影响这里的一个对象，而同样是这个ReportService的其他实例，则完全不受影响。这样做的好处是，我们不必写那么多类，而是根据需要在程序运行时组合出不同的对象。

在这里，相信你再一次意识到了要学习多种程序设计语言的重要性。Java只有类这种组织方式，所以，很多有差异的概念只能用类这一个概念表示出来，思维就会受到限制，而不同的语言则提供了不同的表现形式，让概念更加清晰。

前面只是讲了面向组合编程在思考方式的转变，下面我们再来看设计上的差异。举个例子，我们有个字体类（Font），现在的需求是，字体能够加粗（Bold）、能够有下划线（Underline）、还要支持斜体（Italic），而且这些能力之间是任意组合的。

如果采用继承的方式，那就要有8个类：

普通字体	Font
加粗字体	BoldFont
下划线字体	UnderlineFont
斜体	ItalicFont
加粗下划线字体	BoldUnderlineFont
加粗斜体字体	BoldItalicFont
下划线斜体字体	UnderlineItalicFont
加粗下划线斜体字体	BoldUnderlineItalicFont

而采用组合的方式，我们的字体类（Font）只要有三个独立的维度，也就是是否加粗（Bold）、是否有下划线（Underline）、是否是斜体（Italic）。这还不是终局，如果再来一种其他的要求，由3种要求变成4种，采用继承的方式，类的数量就会膨胀到16个类，而组合的方式只需要再增加一个维度就好。我们把一个M*N的问题，通过设计转变成了M+N的问题，复杂度的差别一望便知。

虽然我们一直在说，Java在面向组合编程方面能力比较弱，但Java社区也在尝试不同的方式。早期的尝试有[Q14j](#)，后来Java 8加入了default method，在一定程度上也可以支持面向组合的编程。这里我们只是讲了面向对象社区在组合方面的探索，后面讲函数式编程时，还会讲到函数式编程在这方面的探索。

总结时刻

今天，我们学习了面向对象的第二个特点：继承。继承分为两种，实现继承和接口继承。实现继承是站在子类的视角看问题，接口继承则是站在父类的视角。

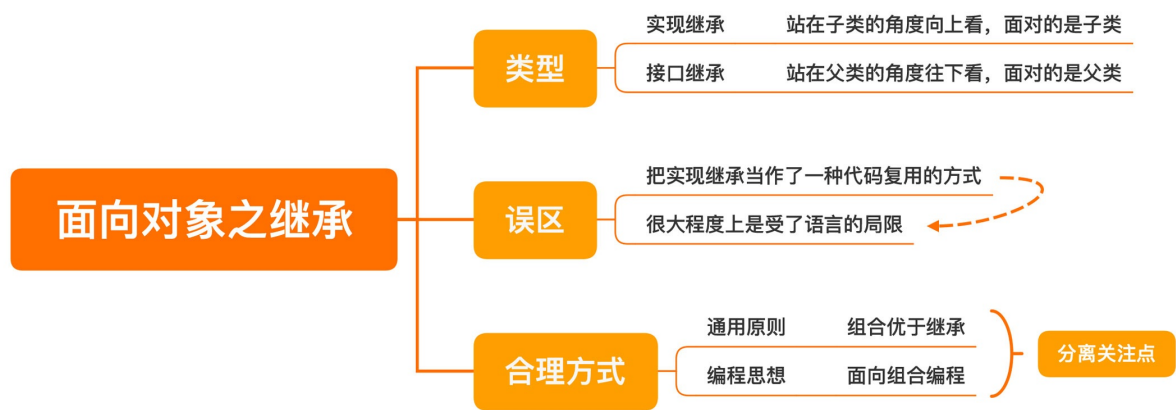
很多程序员把实现继承当作了一种代码复用的方式，但实际上，实现继承并不是一个好的代码复用的方式，之所以这种方式很常见，很大程度上是受了语言的局限。

Ruby的mixin机制，Scala提供的trait以及C++提供的私有继承都是代码复用的方式。即便只使用Java，也可以通过组合而非继承的方式进行代码复用。

今天我们还讲到这些复用方式背后的编程思想：面向组合编程。它给我们提供了一个不同的视角，但支撑面向组合编程的是分离关注点。将不同的关注点分离出来，每一个关注点成为一个模块，在需要的时候组装起来。面向组合编程，在设计本身有很多优秀的地方，可以降低程序的复杂度，更是思维上的转变。

现在你已经知道了，在继承树上从下往上看，并不是一个好的思考方式，那从上往下看呢？下一讲，我们就来讲讲继承的另外一个方向，接口继承，也就是面向对象的第三个特点：多态。

如果今天的内容你只能记住一件事，那请记住：**组合优于继承**。



思考题

最后,我想请你去了解一下一种叫[DCI \(Data, Context和 Interaction\)](#)的编程思想,结合今天的课程,分享一下你对DCI的理解。欢迎在留言区分享你的想法。

感谢阅读,如果你觉得这一讲的内容对你有帮助的话,也欢迎把它分享给你的朋友。

精选留言:

- Jxin 2020-06-29 11:13:12
 - 1.链接打不开,应该要翻墙,回家再看。
 - 2.DCI和小类大对象的理念,在实现手法上很像。
 - 3.先说看好的点,DCI模式在单一职责上能做到更好(ddd的充血模型很容易肿成上帝对象)。一个data在不同的context具有不同的interface方法,这样的划分,在隔离变化(调用方依赖抽象接口实现功能)和复杂性隔离(只关心当前context需要关心的行为)都又不错的效果。
 - 4.再说不看好的,DCI的数据和行为分离不敢苟同。是先有行为再有数据还是先有数据再有行为,这是一个问题。但可以确定的是,行为和数据的相关是密切的,以至于傻傻分不清楚。所以我还是偏向于,抽象模型时,行为和数据应该是一个整体。
 - 5.总结,DCI还是扩展了我的认知。DDD在封装充血模型时有一个很蛋疼的点,为了复用部分属性和行为,而为充血模型加入了新的属性和行为。多次迭代后,充血模型就很容易肿大,边界模糊(涵盖的面广了也就模糊了)。而DCI和小类大对象的思考方式很好的做了补充。大致都是在特定的context里,用不同的小类组成特定的领域对象,既能做到复用也能隔离复杂性。
 - 6.C++有多继承的方式来实现这种组合模式。但java语法特性上就不支持。那么无论是小类大对象还是DCI都无法用合适的代码结构来描述(我认为,哪怕用回调函数+接口实现的写法,也不能很好描述这个语义)。栏主觉得java该怎么办? [2赞]

- 阳仔 2020-06-29 09:13:15

继承是面向对象的基本原则之一,但在编码实践中能用组合来实现尽量使用组合。
DCI也是一种编码规范,它是对面向对象编程的一种补充,其实核心思想也是关注点分离

作者回复2020-06-29 13:48:37
很好的理解!

- Cc°° 2020-06-29 09:04:48

把一个 $M \times N$ 的问题,通过设计转变成了 $M+N$ 的问题。
这个应该是把 2^N 问题变为了 N 的问题吧?

作者回复2020-06-29 13:48:18

换了一个角度看，也是。

- Julien 2020-06-29 07:00:16

```
MODULE(ProductFetcher) {  
...  
}
```

```
MODULE(ReportGenerator) {  
...  
}
```

```
class ReportService:  
INCLUDE(ProductFetcher),  
INCLUDE(ReportGenerator) {  
...  
}
```

这段代码，为什么不用组合呢？

作者回复2020-06-29 11:01:37

你说的字段的组合方式吧，可以用。但如果有更有语义的表示方式，选择更好的表示方式是一个更好的选择。

- NIU 2020-06-29 06:31:34

所谓组合就是类的引用吧，面向对象编程时，如果不是继承关系，那大概率就是引用类来实现功能的组合吧

作者回复2020-06-29 11:59:36

Java 的类引用可以表达很多概念，属性和组合都是通过同样的概念表示出来的。好处就是简单，坏处就不清楚。